

Dynamic programming algorithms, efficient solution of the LP-relaxation and approximation schemes for the Penalized Knapsack Problem

Federico Della Croce^{a,b}, Ulrich Pferschy^c, Rosario Scatamacchia^{a,*}

^a*Dipartimento di Automatica e Informatica, Politecnico di Torino,
Corso Duca degli Abruzzi 24, 10129 Torino, Italy,
{federico.dellacroce, rosario.scatamacchia}@polito.it*
^b*CNR, IEIIT, Torino, Italy*

^c*Department of Statistics and Operations Research, University of Graz,
Universitaetsstrasse 15, 8010 Graz, Austria,
pferschy@uni-graz.at*

Abstract

We consider the 0–1 Penalized Knapsack Problem (PKP). Each item has a profit, a weight and a penalty and the goal is to maximize the sum of the profits minus the greatest penalty value of the items included in a solution. We propose an exact approach relying on a procedure which narrows the relevant range of penalties, on the identification of a core problem and on dynamic programming. The proposed approach turns out to be very effective in solving hard instances of PKP and compares favorably both to commercial solver CPLEX 12.5 applied to the ILP formulation of the problem and to the best available exact algorithm in the literature. Then we present a general inapproximability result and investigate several relevant special cases which permit fully polynomial time approximation schemes (FPTASs).

Keywords: Penalized Knapsack problem, Exact algorithm, Dynamic Programming, Approximation Schemes

1. Introduction

We consider the 0–1 Penalized Knapsack Problem (PKP), as introduced in [1]. PKP is a generalization of the classical 0–1 Knapsack Problem (KP) ([7], [5], [3]), where each item has a profit, a weight and a penalty. The problem calls for maximizing the sum of the profits minus the greatest penalty value of the items included in a solution.

*Corresponding author: Rosario Scatamacchia

PKP has applications in resource allocation problems with a bi-objective function involving the maximization of the sum of profits against the minimization of the maximum value of a feature of interest. As an example, applications arise in batch production systems where the processing time/cost of batches of products depends on the maximum processing time of each product. PKP also occurs as sub-problem within algorithmic frameworks designed for more complex problems. For instance, PKP arises as a pricing sub-problem in branch-and-price algorithms for the two-dimensional level strip packing problem in [4].

PKP is \mathcal{NP} -hard in the weak sense since it contains the standard KP as special case, namely when the penalties of the items are equal to zero, and it can be solved by a pseudo-polynomial algorithm. In [1], it is mentioned that a dynamic programming approach with time complexity $O(n^2c)$ can be easily determined, with n and c being the number of items and the capacity of the knapsack respectively. Also, an exact algorithm is presented and successfully tested on instances with 1000 variables while running into difficulties on instances with 10000 variables.

In this work we propose an exact approach that relies on a procedure narrowing the relevant range of penalties and on dynamic programming. First, a straightforward pseudo-polynomial algorithm running with complexity $O(\max\{n \log n, nc\})$ is lined out. Then, as our major contribution, we devise a dynamic programming algorithm based on a core problem and the algorithmic framework proposed in [8]. We investigate the effectiveness of our approach on a large set of instances generated according to the literature and involving different types of correlations between profits, weights and penalties. The proposed approach turns out to be very effective in solving hard instances and compares favorably to both solver CPLEX 12.5 and the exact algorithm in [1], successfully solving all instances with up to 10000 items.

Finally, we provide additional theoretical insights into the problem, in particular about upper bounds. We derive a surprising negative approximation result, but also investigate several relevant special cases which still permit fully polynomial time approximation schemes (FPTAS).

The paper is organized as follows. In Section 2, the linear programming formulation of the problem is introduced. In Section 3, we provide some insights on structure and properties of PKP. We outline the proposed exact solution approach in Section 4 and discuss the computational results in Section 5. Approximation results are presented in Section 6. Section 7 concludes the paper with some remarks. Note that this report is a slightly extended version of a journal paper.

2. Notation and problem formulation

In PKP a set of n items is given together with a knapsack with capacity c . Each item j has a weight w_j , a profit p_j and a penalty π_j . We will assume that all data are non-negative integer values. Note that [1] assume only the integrality of weights. Similar to their work, all our algorithmic contributions do

not require integral profits or penalties. However, the theoretical approximation results of Section 6 partially rely on integral values. We will assume that items are sorted in decreasing order of penalties, i.e.

$$\pi_1 \geq \pi_2 \geq \dots \geq \pi_n. \quad (1)$$

The problem calls for maximizing the total profit minus the greatest penalty value of the selected items without exceeding the knapsack capacity c . In order to derive an ILP-formulation, we associate with each item j a binary variable x_j such that $x_j = 1$ iff item j is placed into the knapsack. Also, we associate a real variable Π with the decrease in the objective produced by the highest penalty value of the items placed in the knapsack. The problem can be formulated as follows:

$$\text{(PKP)} \quad \text{maximize} \quad \sum_{j=1}^n p_j x_j - \Pi \quad (2)$$

$$\text{subject to} \quad \sum_{j=1}^n w_j x_j \leq c \quad (3)$$

$$\pi_j x_j \leq \Pi \quad j = 1, \dots, n \quad (4)$$

$$x_j \in \{0, 1\} \quad j = 1, \dots, n \quad (5)$$

$$\Pi \in \mathbb{R} \quad (6)$$

(3) is the standard capacity constraint. Constraints (4) ensure that Π will carry the highest penalty value in any feasible solution of PKP; variable Π can be defined in (6) as real and will always attain one of the values π_j in an optimal solution. The objective function (2) maximizes the sum of the profits minus the greatest penalty value of the selected items. Its optimal function value will be denoted by z^* . For any considered sub-problem PP the optimal objective function value will be written as $z(PP)$. The item yielding the optimal penalty will be called the *leading item* and is denoted by j^* .

For further analysis, we will consider the penalty value Π as a fixed parameter and define $PKP(\Pi)$ as an instance of PKP where the penalty value in the objective function is fixed to Π . This simply means that all items j with $\pi_j > \Pi$ are eliminated from consideration and the remaining problem reduces to a standard 0–1 knapsack problem. Obviously, $PKP(\Pi)$ only needs to be considered for the n relevant choices $\Pi \in \{\pi_1, \dots, \pi_n\}$. This implies that PKP can be solved to optimality by solving at most n KP instances of type $PKP(\Pi)$ and taking the maximum objective function value.

3. Generalities and algorithmic insights

We discuss here structure and properties of PKP. We provide a characterization of the linear relaxation of PKP and propose an improved variant of the procedure proposed in [1] for computing upper bounds on the sub-problems induced by the selection of the leading item. Then, we outline a basic dynamic programming algorithm with running time $O(\max\{n \log n, nc\})$.

3.1. Computing upper bounds

A natural upper bound on PKP is given by the optimal solution value of the linear relaxation of the problem, denoted as PKP^{LP} , where constraints (5) are replaced by $x_j \in [0, 1]$, i.e. items can be split and only a fractional part is packed. In this case a proportional part of the penalty applies. The optimal objective function value will be denoted by z^{LP} . The LP-relaxation parametrically depending on Π will be denoted as $PKP^{LP}(\Pi)$ with optimal solution value $z^{LP}(\Pi)$. In the LP-relaxation for a given value Π , each variable x_j is upper bounded by the following expression, since items with penalty exceeding Π are reduced by scaling:

$$x_j \leq \min\{1, \Pi/\pi_j\}. \quad (7)$$

After imposing this bound the problem reduces again to an instance of KP (for fixed Π) for which the LP-relaxation is trivial. We can give the following characterization for $PKP^{LP}(\Pi)$.

Theorem 1. $z^{LP}(\Pi)$ is a piecewise-linear concave function in Π consisting of at most $2n$ linear segments.

Proof. Consider an arbitrary value of Π and the corresponding solution $x^{LP}(\Pi)$. Let S denote the set of items j with $x_j^{LP}(\Pi) = \Pi/\pi_j$, i.e. all items whose values are currently bounded by the considered penalty value Π . The current split item will be denoted as s .

We analyze the slope on the left-hand side of $(\Pi, z^{LP}(\Pi))$ by considering the change of the function implied by a decrease of the penalty bound from Π to $\Pi - \varepsilon$ for some small $\varepsilon > 0$. Formally, this change $\delta(\Pi)$ is given as follows:

$$\begin{aligned} \delta(\Pi) &= z^{LP}(\Pi) - z^{LP}(\Pi - \varepsilon) = -\varepsilon + \underbrace{\sum_{j \in S} \frac{\varepsilon}{\pi_j} p_j}_{\text{reduction of items in } S} - \underbrace{\frac{p_s}{w_s} \sum_{j \in S} \frac{\varepsilon}{\pi_j} w_j}_{\text{increase of split item}} \\ &= \varepsilon \left(-1 + \sum_{j \in S} \frac{w_j}{\pi_j} \underbrace{\left(\frac{p_j}{w_j} - \frac{p_s}{w_s} \right)}_{\geq 0} \right) \end{aligned} \quad (8)$$

This expression can be positive or negative, but it shows that $\delta(\Pi)$ is proportional to ε . Thus, in a neighborhood of $(\Pi, z^{LP}(\Pi))$ the function consists of a linear piece which will end in one of the following three cases:

1. $\Pi - \varepsilon = \pi_k$ for some $k \notin S$. This means that lowering Π , a new item is found for inclusion in S . Plugging in the extended set S in (8) will clearly increase the change $\delta(\Pi)$.
2. x_s reaches 1. This means that the split item becomes integral and item $s + 1$ becomes the new split item. Thus, we replace $\frac{p_s}{w_s}$ by $\frac{p_{s+1}}{w_{s+1}}$ in (8) again implying an increase of $\delta(\Pi)$.

3. x_s reaches $(\Pi - \varepsilon)/\pi_s$. This means that the increase of the split item reaches the lowered penalty bound. In this case, s is included in S and $s + 1$ becomes the new split item. Again, $\delta(\Pi)$ is increased by combining both of the above arguments.

Summarizing, we have shown that starting with an arbitrary value of Π and decreasing Π , $z^{LP}(\Pi)$ consists of a linear piece which ends with some $\Pi' \leq \Pi$ in one of three possible configurations. The slope of this linear piece is given by $\frac{\delta(\Pi)}{\varepsilon}$. The preceding linear segment of $z^{LP}(\Pi')$ will have an *increased* change $\delta(\Pi')$ if Π' is further decreased. This means that the previous linear segment at $z^{LP}(\Pi')$ has a *larger slope* than $z^{LP}(\Pi)$. Thus, the slope of $z^{LP}(\Pi)$ is *decreasing* with increasing Π which proves the concavity of $z^{LP}(\Pi)$.

Starting the above procedure with $\Pi = \max_{j=1}^n \pi_j$ and reducing Π iteratively until $\Pi = 0$, it is clear that each item may cause the end of a linear segment of $z^{LP}(\Pi)$ at most twice: Once, by becoming a new split item and once by being included in set S . Each such event can occur at most once for each item. Therefore, there can be at most $2n$ linear pieces. \square

Exploiting this characterization one can easily construct a solution algorithm based on binary search over the penalty space with $O(n \log \pi_1)$ time. Some more effort is necessary to reach a binary search only over the n relevant penalty values, which yields the following proposition.

Proposition 2. *PKP^{LP} can be solved in $O(n \log n)$ time.*

Proof. Algorithmically, one can easily exploit the structure established in Theorem 1 by performing a binary search over Π to determine a maximum¹ of the concave function $z^{LP}(\Pi)$. For each query value Π , one can compute the split item in linear time and also assemble the corresponding set S in one pass through the set of items. Thus, for each query value Π the sign of the slope can be calculated from (8) in linear time.

Applying the binary search over all possible penalty values would yield a total running time of $O(n \log \pi_1)$ which is polynomial in the size of the (binary) encoded input, i.e. weakly polynomial. To obtain a strongly polynomial time algorithm whose running time depends only on the number of input values, we can first perform a binary search over all n values π_j and thus compute in $O(n \log n)$ time the interval of two consecutive penalties $[\pi_k, \pi_{k-1}]$ for some k (with $\pi_{k-1} \geq \pi_k$ according to (1)) which will include the optimal penalty value Π^{LP} .

Let us denote the optimal split item by s^{LP} and the split items associated with penalties π_{k-1} and π_k by s_{k-1} and s_k respectively. If we consider the item sorted by decreasing $\frac{p_i}{w_j}$, item s_{k-1} will precede item s_k in the ordering.

If $\pi_{k-1} = \pi_k$, clearly we have $s^{LP} = s_{k-1} = s_k$ and $\Pi^{LP} = \pi_{k-1} = \pi_k$. Otherwise, we have to find s^{LP} in the interval $[s_{k-1}, s_k]$ and related Π^{LP} in the interval

¹Note that the maximum is not necessarily unique, since there may exist a linear segment of $z^{LP}(\Pi)$ with slope 0.

$[\pi_k, \pi_{k-1}]$.

Let us consider a generic item \bar{s} as candidate for the split item. The best penalty Π associated with it can be calculated as follows. Given the interval $[\pi_k, \pi_{k-1}]$, we set $x_j = 1$ ($j = 1, \dots, \bar{s}-1$) if $\pi_j \leq \pi_k$, $x_j = \frac{\Pi}{\pi_j}$ otherwise. This implies that the weight and profit sums of the items preceding \bar{s} are linear functions of Π in the form

$$\sum_{j=1}^{\bar{s}-1} w_j x_j = \gamma_1 \Pi + \gamma_2, \quad (9)$$

$$\sum_{j=1}^{\bar{s}-1} p_j x_j = \theta_1 \Pi + \theta_2, \quad (10)$$

with non-negative coefficients $\gamma_1, \gamma_2, \theta_1, \theta_2$ determined according to the above x_j setting. Item \bar{s} can be the split item if and only if $\sum_{j=1}^{\bar{s}-1} w_j x_j < c$ and its value $x_{\bar{s}}$ fulfills the capacity (i.e. $x_{\bar{s}} = \frac{c - \sum_{j=1}^{\bar{s}-1} w_j x_j}{w_{\bar{s}}}$) while satisfying the bound (7). Correspondingly, the feasible interval of Π , denoted as $I_{\bar{s}}(\Pi)$, which allows \bar{s} to be the split item is defined by the following system of inequalities:

$$I_{\bar{s}}(\Pi) := \begin{cases} \pi_k \leq \Pi \leq \pi_{k-1} \\ \gamma_1 \Pi + \gamma_2 \leq c - \varepsilon \\ \frac{c - \gamma_1 \Pi - \gamma_2}{w_{\bar{s}}} \leq \beta \end{cases} \quad \text{with } \beta = \begin{cases} 1 & \text{if } \pi_{\bar{s}} \leq \pi_k; \\ \frac{\Pi}{\pi_{\bar{s}}} & \text{otherwise} \end{cases} \quad (11)$$

Item \bar{s} is a relevant candidate for the split item only if the corresponding interval $I_{\bar{s}}(\Pi)$ is non-empty. In such a case, the overall profit given by \bar{s} as split item is

$$P_{\bar{s}}(\Pi) = \sum_{j=1}^{\bar{s}-1} p_j x_j + p_{\bar{s}} x_{\bar{s}} - \Pi = (\theta_1 - \frac{p_{\bar{s}}}{w_{\bar{s}}} \gamma_1 - 1) \Pi + \theta_2 + \frac{p_{\bar{s}}}{w_{\bar{s}}} (c - \gamma_2) \quad (12)$$

and will be maximized by choosing either the left extreme of $I_{\bar{s}}(\Pi)$ if the term $(\theta_1 - \frac{p_{\bar{s}}}{w_{\bar{s}}} \gamma_1 - 1) < 0$ or the right extreme otherwise.

Summarizing, the best penalty value associated with a candidate item can be computed in constant time if coefficients $\gamma_1, \gamma_2, \theta_1, \theta_2$ are given. Hence, we may first consider item s_{k-1} as candidate for s^{LP} and compute related coefficients in (9)–(10) and penalty value. Then, we iteratively move to the next item after updating coefficients $\gamma_1, \gamma_2, \theta_1, \theta_2$ in one pass due the inclusion of the previous candidate item among items $j = 1, \dots, \bar{s}-1$. After the evaluation of item s_k , the optimal split item s^{LP} and penalty Π^{LP} are returned. Since the execution time of this part is bounded by $O(n)$, the overall complexity for solving the LP relaxation is $O(n \log n)$. \square

In fact, we can do even better by interleaving a median search for the optimal penalty value with a median search for the split item.

Proposition 3. *PKP^{LP} can be solved in $O(n \log \log n)$ time.*

Proof. We describe an algorithm performing an iterative median search for Π^{LP} . In each step of the algorithm we are given a feasible interval for Π^{LP} denoted as $[\eta^\ell, \eta^u]$. Then we determine the median η^m among all penalty values π_j in this interval. This value η^m is considered as a candidate for Π^{LP} . For the resulting solution it is easy to determine the slope of $z^{LP}(\eta^m)$ and decide accordingly whether η^m is set as new upper or lower bound of the search interval.

To find the split item more efficiently in each iteration we will avoid sorting the items by efficiency but instead employ a linear time algorithm as described in [3, ch. 3.1]. It is based on iterated bisection of the item set into sets of items with higher resp. lower efficiency than a current target value e_i . We will keep a sorted list E consisting of all target values e_i (in decreasing order of efficiencies) considered so far during the search for a split item over the different penalty candidate values.

Then we introduce the following data structures. Set S_i contains all items with a penalty value in the current search interval and efficiency value between e_i and the next larger efficiency target value e_{i-1} in E . Formally,

$$S_i := \{j \in N \mid \eta^\ell \leq \pi_j \leq \eta^u, e_{i-1} > e_j \geq e_i\}.$$

For each set S_i we also define the canonical weight sum $W(S_i) := \sum_{j \in S_i} w_j$.

For the currently considered interval bounds η we use the following auxiliary weight arrays for each $e_i \in E$:

$$W_i(\eta) := \sum_j w_j \text{ over all } j \text{ with } e_j > e_i \text{ and } \pi_j \leq \eta \quad (13)$$

$$R_i(\eta) := \sum_j w_j \text{ over all } j \text{ with } e_j > e_i \text{ and } \pi_j > \eta \quad (14)$$

These arrays allow an easy evaluation of each target value $e_i \in E$ since the total weight of all items with efficiency higher than e_i and reduced to the current penalty bound η is given by $W_i(\eta) + R_i(\eta)$. Taking also the weight of the item with efficiency e_i into account, one can easily determine in constant time for a give target e_i whether the split item is found or a higher resp. lower target value for the efficiency should be considered.

It remains to explain the update of the auxiliary data structures. Whenever a new efficiency target value e_k is generated for some item k with $e_{i-1} \geq e_k \geq e_i$ and inserted into E , the current set S_i is partitioned into two sets, namely S_k and (a new set) S_i with the obvious definition. Also the array entries $W_k(\eta)$ resp. $R_k(\eta)$ are generated from $W_{i-1}(\eta)$ resp. $R_{i-1}(\eta)$, while $W_i(\eta)$ resp. $R_i(\eta)$ remain unchanged. These update operations can be done trivially by considering all items of the original set S_i explicitly.

In each main iteration, a new penalty search value η^m is considered. Therefore, all sets S_i are bipartitioned into two disjoint sets S_i^ℓ and S_i^u with $S_i^\ell \cup S_i^u = S_i$ and $\pi_j \leq \eta^m$ for $j \in S_i^\ell$, resp. $\pi_j > \eta^m$ for $j \in S_i^u$. After deciding on the new search interval one of these two sets will replace S_i . Furthermore,

the auxiliary array entries are generated for η^m . This can be done by setting

$$W_i(\eta^m) := W_i(\eta^\ell) + \sum_{k \leq i} W(S_k^\ell), \quad (15)$$

$$R_i(\eta^m) := R_i(\eta^u) + \sum_{\substack{k \leq i \\ j \in S_k^u}} w_j \frac{\eta^m}{\pi_j}. \quad (16)$$

Concerning the running time, the binary search over all n penalty values, i.e. over all candidates η^m , requires $O(\log n)$ iterations. In each iteration we consider explicitly each item with a penalty value in the current interval $[\eta^\ell, \eta^u]$ to determine the median η^m and to update the auxiliary arrays in (15) and (16). (Note that these array entries also have to be generated if no item in the current interval is involved. However, since $|E| \leq \log^2 n$ there can be at most $\log^3 n$ such entries.) Since the search interval is bisected in each iteration, this sums up to $O(n)$ time.

The effort for finding the split items (and thus the solution of the linear relaxation of the KP implied by the current penalty bound η) depends on the sequence of efficiency target values considered in each iteration. Here, we will exploit the fact that evaluating a target value can be done in constant time employing (13) and (14). If the target value was already considered in an earlier iteration for a previous penalty search value, then it is included in E and no additional effort is required. Note that at most $O(\log^2 n)$ such evaluations take place during the execution of the algorithm. If the target value, say e_k , is considered for the first time, the corresponding set S_k has to be generated as described above from a previously existing set S_i , which requires considering all items in S_i .

In the first iteration (for the median of all penalty values), searching for the split item starts with $O(n)$ time for the first median (over all efficiencies), then another $O(n/2)$ time for the second target value (i.e. the median of the upper or lower half of efficiency values), and so on. Searching for t -th target value will require $O(n/2^{t-1})$ time. This includes also going through all items of the associated set S_i which is of cardinality $n/2^{t-1}$.

The same holds for the second iteration, except for the first target value, since the median of all efficiencies was for sure considered in the first iteration. The second target value may or may not have been considered in the first iteration. Thus, we have to take the corresponding effort of $O(n/2)$ time into account.

In the third iteration, the effort for the second target value is only relevant, if it was not considered in the second iteration. Generalizing this argument over all iterations and taking – for the time being – only the first t efficiency target values into account, it turns out that the effort for deciding the t -th target values in total over all $\log n$ iterations can be at most $O(n)$. This results from considering each of the subsets of $n/2^{t-1}$ items at most once. The total effort of this part is $O(n \cdot t)$.

Continuing the analysis for target values numbered by $t+1, t+2, \dots, \log n$

we can bound the effort for each iteration over one penalty value by

$$n/2^t + n/2^{t+1} + \dots + 1 \approx n/2^{t-1}.$$

This effort arises for all $\log n$ iterations over all penalty search values. Thus, we can summarize the total running time associated to the solutions of the linear relaxations as:

$$n \cdot t + \log n \frac{n}{2^{t-1}}$$

Plugging in $t = \log \log n$ yields $O(n \log \log n + n)$. \square

We may compute more involved upper bounds on PKP as follows. As pointed out above, the optimal solution of PKP is determined by a penalty value Π and a subset of items j with $\pi_j \leq \Pi$. Therefore, we consider sub-problem $PKP_j := PKP(\pi_j)$ for $j = 1, \dots, n$. Recalling (1) each PKP_j is an instance of KP with item set $\{j, j+1, \dots, n\}$ and capacity c where π_j is subtracted from the final solution value.

Fixing $\Pi = \pi_j$ for some j is only relevant for the final solution if item j is actually included in the solution. Hence, as in [1], we also consider sub-problem PKP_j^+ , where item j is packed, a fixed penalty of $\Pi = \pi_j$ is subtracted from the objective function, and for the remainder of the solution a KP is solved with capacity $c - w_j$ and item set $\{j+1, \dots, n\}$.

For both PKP_j resp. PKP_j^+ we consider the LP-relaxation as upper bound denoted by PKP_j^{LP} resp. PKP_j^{+LP} . It is easy to see that

$$z(PKP_j^+) \leq z(PKP_j) \tag{17}$$

$$z^* = \max_{j=1, \dots, n} z(PKP_j) \leq \max_{j=1, \dots, n} z(PKP_j^{LP}) =: UB_{sub} \tag{18}$$

$$z^* = \max_{j=1, \dots, n} z(PKP_j^+) \leq \max_{j=1, \dots, n} z(PKP_j^{+LP}) =: UB_{sub}^+ \tag{19}$$

The following dominance relations exist for the upper bounds UB_{sub} , UB_{sub}^+ and z^{LP} .

Proposition 4. *For any PKP instance, we have that*

$$UB_{sub}^+ \leq UB_{sub} \leq z^{LP} \tag{20}$$

and there are instances where the inequalities are strict.

Proof. Clearly, the restricted feasible domain of UB_{sub}^+ cannot lead to a greater value than UB_{sub} and thus $UB_{sub}^+ \leq UB_{sub}$. Let us denote by j' the item yielding UB_{sub} , i.e. $UB_{sub} = z(PKP_{j'}^{LP})$. Computing $PKP^{LP}(\Pi)$ with $\Pi = \pi_{j'}$ gives a feasible solution for the LP relaxation whose value is less than (or equal to) the optimal value z^{LP} but at least as large as UB_{sub} . The latter holds because *all* items are involved (and bounded according to (7)) in the computation while only items i with $\pi_i \leq \pi_{j'}$ are considered for solving $PKP_{j'}^{LP}$. This implies that $UB_{sub} \leq z^{LP}$.

To show that inequalities in (20) can be strict, consider the following PKP instance with $n = 2$ items, capacity $c = 7$ and the entries:

$$p_1 = 10, w_1 = 5, \pi_1 = 1; \quad p_2 = 6, w_2 = 4, \pi_2 = 2$$

For this instance we have $z^{LP} = 12$, $z(PKP_1^{LP}) = -1 + 10 = 9$, $z(PKP_2^{LP}) = -2 + 10 + \frac{2}{4}6 = 11$, $z(PKP_1^{+LP}) = -1 + 10 = 9$, $z(PKP_2^{+LP}) = -2 + 6 + \frac{3}{5}10 = 10$. Thus, we have:

$$UB_{sub}^+ = 10 < UB_{sub} = 11 < z^{LP} = 12$$

□

Although the three bounds can be computed efficiently and can be expected to be reasonably close to the optimal value in practice, we show a negative result on their deviation from the optimum.

Proposition 5. *There are instances of PKP where the differences $(UB_{sub}^+ - z^*)$, $(UB_{sub} - z^*)$ and $(z^{LP} - z^*)$ are arbitrarily large.*

Proof. Consider the following instance with $n = 2$ items, capacity $c = M$ and the following entries: $p_j = w_j = \frac{M}{2} + 1$ and $\pi_j = \frac{M}{2}$ for $j = 1, 2$. In an optimal solution only one item j is packed and, correspondingly, $z^* = p_j - \pi_j = 1$. Also, it is easy to see that $UB_{sub}^+ = \frac{M}{2}$, which, in combination with (20), shows the claim. □

Algorithmically, it is not difficult to see that all values $z(PKP_j^{LP})$ for $j = 1, \dots, n$ can be computed in $O(n \log n)$ time. Also from a practical point of view, the effort hardly exceeds sorting. As a preprocessing step an auxiliary array is constructed containing all items sorted in decreasing order of efficiencies p_j/w_j . Then the problems PKP_j^{LP} are considered iteratively for $j = 1, \dots, n$, i.e., in decreasing order of penalties π_j . First, PKP_1^{LP} is solved in linear time and the corresponding split item (i.e. the first item not fully packed into the knapsack) is identified. We keep a pointer to this split item in the sorted array of items. Moving to PKP_2^{LP} , we just remove item 1 from the solution and increase the split item, or possibly move to a new split item by shifting the pointer towards items with lower efficiency. All together, after sorting, all values $z(PKP_j^{LP})$ can be determined in linear time by one pass through the sorted array of items.

In [1], the authors presented an $O(n^2)$ procedure to compute all values $z(PKP_j^{+LP})$ for $j = 1, \dots, n$. In the following, we show that in fact $O(n \log n)$ time is sufficient to perform this task.

Proposition 6. *All values $z(PKP_j^{+LP})$ for $j = 1, \dots, n$ can be computed in $O(n \log n)$ time.*

Proof. First, the items are sorted in decreasing order of efficiencies. Based on this sequence we construct an auxiliary data structure consisting of a binary tree as follows: Each item corresponds to a leaf node of the tree. These are nodes at

level 0. A parent node is associated with each pair of consecutive items (with a singleton remaining at the end for n odd) thus yielding other $\lceil \frac{n}{2} \rceil$ nodes in the level 1 of the tree. This process is iterated recursively up the tree, which trivially reaches a height of $O(\log n)$.

In each node v of the tree we store as $W(v)$ (resp. $P(v)$) the sum of weights (resp. profits) of all items corresponding to leaf nodes in the subtree rooted in v . Clearly, such a tree and its additional information can be built in $O(n)$ time.

For any given capacity c' the corresponding split item and also the value of the optimal LP-relaxation can be found in $O(\log n)$ time by starting at the root node and going down towards a leaf node by applying the following rule in every node v with left and right child nodes v^L and v^R :

If $W(v^L) > c'$ then set $v := v^L$.
Otherwise set $v := v^R$ and $c' := c' - W(v^L)$.

The item corresponding to the leaf node reached by this procedure is the split item. The solution value can be reported by keeping track of the $P(v)$ values during the pass through the tree.

In the main iteration of the algorithm we compute $z(PKP_j^{+LP})$ iteratively for $j = 1, \dots, n$ in decreasing order of penalties π_j . First we remove item j permanently from consideration. This means that the leaf node corresponding to j is removed from the tree and all $O(\log n)$ labels $W(v)$ (resp. $P(v)$) on the unique path from this leaf to the root of the tree are updated by subtracting w_j (resp. p_j). Then we solve an LP-relaxation with capacity $c' := c - w_j$ and add $p_j - \pi_j$ to the objective function. All together there are n iterations, each of which requiring $O(\log n)$ time to find the solution of the LP-relaxation and $O(\log n)$ time to update the labels of the binary tree. \square

Note that we might expect a considerable speed-up of the running time $O(n \log n)$ in a practical implementation since the tree loses vertices in each iteration and path contractions can be performed.

3.2. A basic dynamic programming algorithm

As recalled in [1], a straightforward pseudo-polynomial algorithm for PKP consists of solving j standard knapsack problems PKP_j^+ by the classical dynamic programming by weights running in $O(nc)$. The overall complexity is thus $O(n^2c)$. However, we can do much better by evaluating all n subproblems in one run.

Theorem 7. *PKP can be solved with complexity $O(\max\{nc, n \log n\})$.*

Proof. It suffices to consider the items sorted by increasing penalty and to run the dynamic program for KP only once. If we denote by $F_j(d)$ the optimal solution value of the sub-problem of KP consisting of items $1, \dots, j$ and capacity $d \leq c$, the optimal value of any PKP instance is simply given by

$$\max_{j=0, \dots, n-1} \{F_j(c - w_{j+1}) + p_{j+1} - \pi_{j+1}\} \quad (21)$$

That is, we evaluate the choice of item $j + 1$ as leading item just by considering the maximum profit reachable with the previous items in a knapsack with capacity $c - w_{j+1}$. The running time is $O(nc)$ plus the effort for sorting. \square

4. An exact solution approach

4.1. Overview

The DP algorithm of Theorem 7, hereafter denoted as DP_1 , may be appealing whenever the capacity c is of reasonably limited size. However, for KP it is known that more effective than the iterative addition of items are algorithms based on the core problem. Thus, our idea is to exploit the core concept for PKP similarly to the framework of the *Minknap* algorithm [8]. We remark that the presence of penalties compromises in PKP the structure of an optimal solution with respect to a standard KP. This difference would typically affect the performance of an approach based on a core problem. Further, the presence of penalties limits the effectiveness of the classical dominance rule in KP based on the profits and the weights of the states. Anyhow, from a practical perspective it is still beneficial to run a dynamic programming algorithm starting from the split solution of KP and not from scratch. In addition, by narrowing the interval of penalty values which can possibly lead to an optimal solution, the “noise” added by the penalties can be further reduced.

We propose an exact approach involving two main steps. In the first step, we effectively compute an initial feasible solution for the problem and identify the relevant interval of penalties values possibly leading to an optimal solution. In the second step, we run a dynamic programming algorithm with states based on the core concept. In case the first step yields a reduced problem with a reasonably limited input size, we could as well launch the DP_1 algorithm. In the following subsections we describe the steps of the approach whose pseudo code is presented in Algorithm 1.

4.2. Step 1: Computing an initial feasible solution and the relevant interval of penalty values

The approach takes as input four parameters T_1, T_2, T_3, α and starts by solving the standard knapsack problem KP_1 given by disregarding the penalties of the items in PKP (lines 2-3 in Algorithm 1). This problem is solved as follows. Denote the index of the first item in the optimal solution of KP_1 (according to the ordering (1)) by \bar{f} . The corresponding first feasible solution of PKP has objective value $z(KP_1) - \pi_{\bar{f}}$. Similarly to Proposition 2 in [1], the following proposition holds

Proposition 8. *All items $j = 1, \dots, \bar{f} - 1$ can be discarded without loss of optimality.*

Proof. Since $z(KP_1)$ is the optimal solution value, including any item $j = 1, \dots, \bar{f} - 1$ leads to a solution with profits less than (or equal to) $z(KP_1)$ but induces a penalty greater than (or equal to) $\pi_{\bar{f}}$. \square

Algorithm 1 Exact solution approach

1: **Input:** PKP instance, parameters T_1, T_2, T_3, α . ▷ Step 1

2: $KP_1 = \text{PKP without penalties}$;

3: $(\bar{z}, \bar{j}, \bar{f}) \leftarrow \text{ModMinknap}(KP_1)$;

4: Compute $z(PKP_j^{+LP})$ for $j = \bar{f} + 1, \dots, n$;

5: $UB = \max_j z(PKP_j^{+LP})$;

6: **if** $UB \leq \bar{z}$ **then** $z^* = \bar{z}, j^* = \bar{j}$; **return** (z^*, j^*) ; **end if**

7: $k = \arg \max_j z(PKP_j^{+LP})$;

8: $KP_2 = KP_1 \cap (x_j = 0 \mid j = 1, \dots, k-1)$;

9: $(\hat{z}, \hat{j}, \hat{f}) \leftarrow \text{ModMinknap}(KP_2)$;

10: **if** $\hat{z} > \bar{z}$ **then** $\bar{z} = \hat{z}, \bar{j} = \hat{j}$; **end if**

11: $l \leftarrow \text{Apply (22)}$;

12: $r \leftarrow \text{Apply (23)}$;

13: $\Pi_{max} = \pi_l$;

14: $\Pi_{min} = \pi_r$;

15: **if** $[\Pi_{min}, \Pi_{max}] = \emptyset$ **then** $z^* = \bar{z}, j^* = \bar{j}$; **return** (z^*, j^*) ; **end if**

16: $PKP' = \text{PKP} \cap (x_j = 0 \mid j = 1, \dots, l-1; \Pi \geq \Pi_{min})$;

17: $n' = n - l + 1$; ▷ Step 2

18: **if** $n'c \leq T_1$ and $(r - l + 1) \geq T_2$ **then**

19: $(z', j') \leftarrow DP_1(PKP')$;

20: **if** $z' > \bar{z}$ **then** $z^* = z', j^* = j'$; **else** $z^* = \bar{z}, j^* = \bar{j}$; **end if**

21: **else**

22: $(z^*, j^*) \leftarrow DP_2(PKP', \bar{z}, \bar{j}, T_3, \alpha)$;

23: **end if**

24: **return** (z^*, j^*) ;

Thus, if there is more than one optimal solution of KP_1 , we are interested in the solution yielding the lowest penalty value for PKP, i.e. the largest index \bar{j} . This task is easily accomplished by considering a slight variant of *Minknap*, hereafter denoted as *ModMinknap*, which keeps track of all optimal solutions of KP_1 and the corresponding penalty values in PKP. In addition, we can compute PKP solutions during the iterations of *ModMinknap* just by tracking the largest penalty associated to each feasible state. We then take the overall best solution found for PKP. Denote by \bar{z} its value and by \bar{j} the index of the leading item.

We remark that *ModMinknap* is only a heuristic algorithm for PKP since it does not explicitly consider the penalties of the items in the iterations. At the same, it may “stumble” upon good quality solutions of PKP with just a negligible increase of the computational effort required for solving a KP instance.

Then, we compute $z(PKP_j^{+LP})$ for $j = \bar{j}+1, \dots, n$. If the maximum of these values is not superior to \bar{z} , we have already certified an optimal solution for PKP (lines 4–6 in Algorithm 1). Otherwise we greedily consider the index k yielding the maximum $z(PKP_j^{+LP})$ and solve KP_1 without items $j = 1, \dots, k-1$. We update the values of \bar{z} and \bar{j} if an improving solution is found (lines 7–10 in Algorithm 1).

Finally, we compare the values $z(PKP_j^{+LP})$ with the incumbent solution value \bar{z} and narrow the range of possible penalty values that may lead to an optimal solution of PKP. More precisely, we define indices l and r

$$l := \min \{j : z(PKP_j^{+LP}) > \bar{z}\}; \quad (22)$$

$$r := \max \{j : z(PKP_j^{+LP}) > \bar{z}\}. \quad (23)$$

The relevant interval of penalties is thus $[\Pi_{min}, \Pi_{max}]$, with $\Pi_{min} = \pi_r$ and $\Pi_{max} = \pi_l$ (line 11–14 in Algorithm 1). If this interval is empty, the current PKP solution is also optimal and the algorithm terminates. Otherwise we get a reduced PKP with only items $j = l, \dots, n$ and the additional constraint on the penalty value $\Pi \geq \Pi_{min}$. Denote this problem by PKP' and its number of items by n' , i.e. $n' = n - l + 1$ (lines 15–17 in Algorithm 1).

This first step is expected to be fast since it relies on solving two standard knapsack problems at most and on effectively computing upper bounds for sub-problems PKP_j^+ . We remark that this step is also sufficient to compute an optimal solution for a large number of instances considered in the literature.

4.3. Step 2: A core-based dynamic programming algorithm

In this step we propose a core-based dynamic programming algorithm, hereafter denoted as DP_2 , that constitutes a revisiting of *Minknap* algorithm. Notice that, if the size of the reduced problem PKP' is reasonably small and the number of relevant penalties is large, we could otherwise solve PKP' by DP_1 and take the best solution between $z(PKP')$ and \bar{z} . The choice between the algorithms is made by comparing the quantities n'/c and $(r-l+1)$ with the threshold parameters T_1 and T_2 (lines 18–23 in Algorithm 1).

DP_2 algorithm searches in PKP' for solutions better than \bar{z} . Given the sorting of the items $j = 1, \dots, n'$ by decreasing $\frac{p_j}{w_j}$, we define an expanding core as the interval of items $C_{a,b} = \{a, \dots, b\}$ with items a and b as variable extremes. Correspondingly, we define the set of 0–1 partial vectors enumerated within the core as

$$X_{a,b} = \{x_j \in \{0, 1\}, j \in C_{a,b}\}. \quad (24)$$

Since in any iteration of the algorithm we will have the following situation

$$\overbrace{x_1, \dots, x_{a-1}}^{x_j = 1}, C_{a,b}, \overbrace{x_{b+1}, \dots, x_{n'}}^{x_j = 0} \quad (25)$$

we associate each partial vector $\tilde{x} \in X_{a,b}$ with a state $(\tilde{\nu}, \tilde{\mu}, \tilde{\pi}_{core}, \tilde{\pi}_{tot})$ where:

1. $\tilde{\nu} = \sum_{j=1}^{a-1} p_j + \sum_{j=a}^b p_j \tilde{x}_j;$
2. $\tilde{\mu} = \sum_{j=1}^{a-1} w_j + \sum_{j=a}^b w_j \tilde{x}_j;$
3. $\tilde{\pi}_{core} = \max_{j=a, \dots, b} \{\pi_j : \tilde{x}_j = 1\};$
4. $\tilde{\pi}_{tot} = \max\{\tilde{\pi}_{core}, \max_{j=1, \dots, a-1} \pi_j\}.$

$\tilde{\nu}$ and $\tilde{\mu}$ are the profits and weights of a solution with variables in the core and all variables to the left of the core; $\tilde{\pi}_{core}$ represents the maximum penalty of the items selected in the core while $\tilde{\pi}_{tot}$ is the overall maximum penalty of the state. Each state with $\tilde{\mu} \leq c$ and $\tilde{\pi}_{tot} \geq \Pi_{min}$ represents a feasible solution of PKP' with value $\tilde{\nu} - \tilde{\pi}_{tot}$. We can now sketch the main steps of DP_2 in the following pseudo code. The algorithm takes as input PKP' , the current solution (\bar{z}, \bar{j}) and parameters T_3, α .

We first sort the items of PKP' by decreasing $\frac{p_j}{w_j}$ and find the split item s' of the standard knapsack problem (KP') induced by disregarding the penalties in PKP' . We then initialize the core with item s' only (lines 1–4 of the pseudo code). Then, we enlarge the core as in *Minknap* (while-loop in lines 6–23) by alternately evaluating the removal of an item a from the left (lines 7–14) and the insertion of an item b from the right (lines 15–22). The expansion of the core is performed by a dynamic programming with states through a procedure, denoted as *Merge*, which iteratively yields undominated states in the enlarged set $X_{a,b} = X_{a+1,b} + a$ or $X_{a,b} = X_{a,b-1} + b$. We may update the current solution (\bar{z}, \bar{j}) if an improved solution is found while enumerating the core (lines 11 and 19).

The dynamic programming with states is combined with an upper bound test to reduce the number of states (lines 5, 12 and 20) and two upper bound tests to limit the insertion of the variables in the core (lines 8–9 and 16–17). The algorithm terminates whenever either the number of states is 0 or all variables have been enumerated in the core. The ingredients of the algorithm are detailed in the following.

Algorithm 2 $DP_2(PKP', \bar{z}, \bar{j}, T_3, \alpha)$

```

1: Sort items in  $PKP'$  by decreasing  $\frac{p_i}{w_j}$ ;
2:  $KP' = PKP'$  without penalties;
3: Find the split item  $s'$  of  $KP'$ ;
4:  $a = b = s'$ ,  $C_{a,b} = \{s'\}$ ;  $X_{a,b} = \{(0), (1)\}$ ;
5: Reduce set  $X_{a,b}$ ;
6: while  $X_{a,b} \neq \emptyset$  and  $(b - a + 1 < n')$  do
7:    $a \leftarrow a - 1$ ;
8:   if  $u_0^a > \bar{z}$  then
9:     if  $\tilde{u}^a > \bar{z}$  then
10:       $X_{a,b} \leftarrow Merge(a, X_{a+1,b}, \Pi_{min}, T_3, \alpha)$ ;
11:      Update  $(\bar{z}, \bar{j})$ ;
12:      Reduce set  $X_{a,b}$ ;
13:     end if
14:   end if
15:    $b \leftarrow b + 1$ ;
16:   if  $u_1^b > \bar{z}$  then
17:     if  $\tilde{u}^b > \bar{z}$  then
18:       $X_{a,b} \leftarrow Merge(b, X_{a,b-1}, \Pi_{min}, T_3, \alpha)$ ;
19:      Update  $(\bar{z}, \bar{j})$ ;
20:      Reduce set  $X_{a,b}$ ;
21:     end if
22:   end if
23: end while
24: return  $(\bar{z}, \bar{j})$ ;

```

4.3.1. Dynamic programming with states

The *Merge* procedure performs the enumeration of the variables in the core by resembling the procedure introduced in *Minknap* [8], which in turn corresponds to the recursions of the *primal-dual* dynamic programming algorithm in [9]. The proposed procedure merges, in any iteration, the current set of states X and $X + d$, where $X + d$ is set X with profits, weights and penalties of the states updated according to the removal/insertion of item d from/in the knapsack. In the merging operation the states are kept ordered by increasing weights so as to effectively apply a dominance rule for PKP.

The classical dominance rule in KP considers the weights and profits of the states. For PKP, let us define the quantity $\rho = \nu - \max\{\pi_{core}, \Pi_{min}\}$ which represents the difference between the profit of a state and the minimum penalty that the state must have for yielding an optimal solution. This penalty corresponds to the maximum between Π_{min} and π_{core} since, due to the enumeration of the core, for any state π_{core} constitutes a minimum penalty value in all states originating from it while Π_{min} is the minimum penalty required in any solution with a value greater than \bar{z} . We introduce the following dominance rule for two generic states i and j .

Proposition 9. *Given states i and j and their quantities fulfilling*

$$\mu^i \leq \mu^j, \quad \nu^i \geq \nu^j, \quad \rho^i \geq \rho^j. \quad (26)$$

Then state j is said to be dominated by state i and can be discarded in the search for an optimal solution of PKP.

Proof. The first two conditions represent the dominance of state i in the standard KP. The condition $\rho^i \geq \rho^j$ implies that all successive states deriving from state i and possibly optimal for PKP (i.e. with a penalty greater than Π_{min}) would have a no worse solution value than those deriving from state j . \square

We remark that, given the presence of penalties, the ordering of states by increasing weights does not imply the ordering of the profits as in *Minknap*. To better detect situations of dominance, we apply the rule in Proposition 9 by comparing each state with a number of states (with a lower weight) given by parameter α .

Whenever only the condition involving the penalties prevents the fathoming of state j , we may combine the dominance rule with an upper bound on state j depending on a penalty value $\pi > \max\{\pi_{core}^j, \Pi_{min}\}$.

This upper bound, denoted by $UB(\pi)^j$, is computed as follows. We first solve the linear relaxation of the KP induced by packing the items selected in the core for state j and by disregarding the items outside the core with a higher penalty than π . From the optimal solution value of this problem we then subtract the maximum value between π_{core}^j and Π_{min} . The following proposition holds

Proposition 10. *Given two states i and j and the quantities*

$$\mu^i \leq \mu^j, \quad \nu^i \geq \nu^j, \quad \rho^i < \rho^j, \quad (27)$$

consider the maximum penalty $\hat{\pi}$ which would not induce the dominance of state i according to (26), i.e. $\hat{\pi} = \max \{\pi : \nu^j - \pi > \rho^i\}$. If $UB(\hat{\pi})^j \leq \bar{z}$, then state j can be discarded.

Proof. We analyze the solution values deriving from state j when the overall maximum penalty is upper bounded by a quantity π' . For any $\pi' \leq \hat{\pi}$, since $UB(\hat{\pi})^j \leq \bar{z}$ we can discard state j because all states deriving from state j cannot reach a solution values greater than \bar{z} . Likewise, we can as well discard state j if $\pi' > \hat{\pi}$ since this condition would induce a dominance of state i . \square

Computing $UB(\pi)$ has complexity $O(n)$ and would be time-consuming if the number of states involved is sufficiently large. Thus, we calculate this bound only if the number of states exceeds the threshold value T_3 .

4.3.2. Reduction of the states

To further reduce the set of states, we also perform an upper bound test in constant time for each state. In any iteration, we compute the following upper bound for a state i associated with $X_{a,b}$:

$$UB^i = \begin{cases} \rho^i + (c - \mu^i) \frac{p_{b+1}}{w_{b+1}} & \text{if } \mu^i \leq c \\ \rho^i + (c - \mu^i) \frac{p_{a-1}}{w_{a-1}} & \text{if } \mu^i > c \end{cases} \quad (28)$$

and discard state i if $UB^i \leq \bar{z}$. These upper bounds are computed by replacing the integrality constraint on x_{a-1} and x_{b+1} with $x_{a-1} \geq 0$ and $x_{b+1} \geq 0$ and by disregarding the penalty values of the variables outside the core.

4.3.3. Upper bound tests on the variables outside the core

Since the insertion of variables in the core may be computationally expensive, we perform two upper bound tests whenever an item j is candidate to be included in the core.

We first compute similar bounds to the ones proposed in [2] for KP. Let us denote by u_0^j an upper bound on PKP' without item j . Also, let us denote by u_1^j the upper bound when item j is packed. The following bounds are computed in constant time for each item j :

$$u_0^j = p' - p_j - \Pi_{min} + (c - w' + w_j) \frac{p_{s'}}{w_{s'}} \quad j = 1, \dots, s' - 1 \quad (29)$$

$$u_1^j = p' + p_j - \max\{\pi_j, \Pi_{min}\} + (c - w' - w_j) \frac{p_{s'}}{w_{s'}} \quad j = s' + 1, \dots, n \quad (30)$$

Here $w' = \sum_{j=1}^{s'-1} w_j$ and $p' = \sum_{j=1}^{s'-1} p_j$ represent the weight and the profit of the split solution of KP' . If u_0^j (resp. u_1^j) $\leq \bar{z}$, we can fix variable $x_j = 1$ (resp. $x_j = 0$).

In cascade, we may perform a second test by computing a stronger upper bound in linear time with the number of states. As in *Minknap*, we evaluate

the impacts of removing (inserting) item j with $j < s'$ ($j > s'$) in all states in the current set X , namely we derive states $i \in X + j$ and compute upper bounds (28) on these states. A valid upper bound for item j , denoted as \tilde{u}^j , is constituted by the maximum of these bounds. As pointed out in [8], \tilde{u}^j can be seen as a generalization of the enumerative bound in [6]. If $\tilde{u}^j \leq \bar{z}$, then variable x_j is fixed to the value taken in the split solution.

After this second step, the optimal solution value z^* and the optimal leading item j^* are returned. The optimal solution set of items can be determined by solving the standard knapsack problem $PKP_{j^*}^+$.

5. Computational results

All tests were performed on an Intel i7 CPU @ 2.4 GHz with 8 GB of RAM. The code was implemented in the C++ programming language. We generated the instances according to the generation scheme proposed in [1]. We considered two types of weights: $a1$ and $a2$. In the former, the weights are randomly distributed in $[1, R]$, with R being an arbitrary parameter. In the latter, the weights are equal to $\frac{R}{2} + v$, with v uniformly distributed in $[0, \frac{R}{2}]$. Basically, small weights are not considered in $a2$.

We generated 8 classes of penalties ($\pi1, \dots, \pi8$) and 7 classes of profits ($p1, \dots, p7$) according to different correlations of penalties/profits with the weights, as illustrated in Table 1. The first 6 correlations correspond to classical correlations in KP instances. In class $\pi7$ penalties π_j are equal to $R - w_j + 1$ (constant perimeter correlation) while in class $\pi8$ we set $\pi_j = \frac{R}{w_j}$ (constant area correlation). In class $p7$ we set $p_j = \pi_j w_j$. Finally, three different values of the ratio τ between the knapsack capacity and the sum of the weights of the items are considered: 0.5, 0.1 and 0.01.

π type	Correlation	p type
$\pi1$	No correlation	$p1$
$\pi2$	Weak correlation	$p2$
$\pi3$	Strong correlation	$p3$
$\pi4$	Inverse strong correlation	$p4$
$\pi5$	Almost strong correlation	$p5$
$\pi6$	Subset-sum correlation	$p6$
$\pi7$	Constant perimeter	
$\pi8$	Constant area	
	Profit = area	$p7$

Table 1: Correlation types from [1].

We first generated instances with 1000 items and $R = 1000$. Within each category, five instances were tested for a total of 1680 instances. Similarly, we generated 1680 instances with 10000 items and $R = 10000$. We compared the solutions reached by the proposed exact approach, the algorithm in [1] and CPLEX 12.5 running on model (PKP). After some preliminary test runs, we

chose the following parameter values for our approach: $\alpha = 15$, $T_1 = 5 * 10^9$, $T_2 = \frac{n}{10}$, $T_3 = 3 * 10^6$. The parameters of the ILP solver were set to their default values.

The results are summarized in Tables 2 and 3 in terms of average, maximum CPU time and number of optima obtained within a time limit of 100 seconds. The average CPU times consider also the cases where the time limit is reached. The results are aggregated by profit classes and weight types. Each entry in the tables reports the results over 120 instances. Detailed results for all correlations and capacity ratios are given at the end of this section.

$n = 1000$		CPLEX 12.5			Algorithm in [1]			Proposed exact approach		
Profit class	Weight type	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
$p1$	$a1$	0.18	0.25	120	0.00	0.01	120	0.00	0.00	120
	$a2$	0.19	0.48	120	0.00	0.01	120	0.00	0.03	120
$p2$	$a1$	0.39	1.69	120	0.00	0.01	120	0.00	0.10	120
	$a2$	1.12	6.32	120	0.00	0.01	120	0.01	0.27	120
$p3$	$a1$	3.90	100.00	117	0.04	0.89	120	0.01	0.40	120
	$a2$	6.83	100.00	117	0.50	8.02	120	0.02	0.29	120
$p4$	$a1$	59.56	100.00	57	0.10	1.38	120	0.02	0.18	120
	$a2$	66.97	100.00	46	0.28	7.75	120	0.04	0.26	120
$p5$	$a1$	4.13	100.00	117	0.03	0.89	120	0.02	0.20	120
	$a2$	14.97	100.00	114	0.41	4.50	120	0.07	1.40	120
$p6$	$a1$	2.67	90.38	120	0.00	0.06	120	0.00	0.03	120
	$a2$	2.97	13.57	120	0.01	0.15	120	0.01	0.08	120
$p7$	$a1$	27.58	100.00	89	0.00	0.01	120	0.00	0.02	120
	$a2$	38.24	100.00	75	0.01	0.06	120	0.01	0.07	120

Table 2: Summary results for instances with 1000 items and different correlations between profits and weights: time (s) and number of optima over 120 instances.

$n = 10000$		CPLEX 12.5			Algorithm in [1]			Proposed exact approach		
Profit class	Weight type	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
$p1$	$a1$	0.95	2.68	120	0.01	0.02	120	0.01	0.03	120
	$a2$	4.19	100.00	117	0.01	0.04	120	0.01	0.03	120
$p2$	$a1$	5.41	33.84	120	0.01	0.02	120	0.02	0.13	120
	$a2$	12.43	100.00	114	0.01	0.07	120	0.04	0.73	120
$p3$	$a1$	44.61	100.00	74	25.59	100.00	96	2.59	58.46	120
	$a2$	74.13	100.00	46	48.26	100.00	74	5.53	29.00	120
$p4$	$a1$	91.60	100.00	11	17.68	100.00	106	2.81	18.88	120
	$a2$	94.69	100.00	7	26.34	100.00	106	7.82	80.02	120
$p5$	$a1$	25.45	100.00	95	10.70	100.00	113	2.66	70.71	120
	$a2$	65.99	100.00	48	23.74	100.00	101	7.04	58.60	120
$p6$	$a1$	83.08	100.00	58	0.67	40.17	120	0.22	5.38	120
	$a2$	81.05	100.00	44	3.62	100.00	119	1.95	16.65	120
$p7$	$a1$	51.00	100.00	63	0.17	0.94	120	0.42	2.50	120
	$a2$	40.12	100.00	75	0.54	3.94	120	1.41	11.46	120

Table 3: Summary results for instances with 10000 items and different correlations between profits and weights: time (s) and number of optima over 120 instances.

From Tables 2 and 3 we see that, for the instances with 1000 items, both the proposed exact approach and the algorithm in [1] outperform CPLEX 12.5 which does not reach all the optima within the time limit. Although the performances of the algorithms are similar, we note that our approach generally performs slightly better and requires 1.4 seconds at most for solving to optimality all instances.

In the largest instances with 10000 items, our algorithm strongly outperforms both CPLEX 12.5 and the algorithm in [1]. Our approach is capable of reaching all optima with limited CPU time (80 seconds at most for an instance in class

$p4$) while the solver and the competing algorithm run out of time for several large instances. The largest differences in computational times involve instances in classes $p3$, $p4$ and $p5$.

The most challenging instances for our algorithm turned out to be the ones without small weights ($a2$). In general, the absence of small weights might increase the computational effort required for solving even standard KP instances (as pointed out, e.g., in [1]) and this is presumably the reason of the increase in CPU times of our algorithm as well.

In many instances, the first main step relying on solving standard KPs is sufficient to certificate an optimal solution for PKP. Indeed, this constitutes a remarkable strength of our procedures. In Tables 4 and 5 we report the percentage of the optimal solutions already computed by the first step of the procedure for the instances with 1000 and 10000 items respectively. Averaged computational times (% of the total CPU time) of the two steps of our approach are also reported. Finally, we report the average and maximum values (in thousands) of the maximum number of states reached by DP_2 algorithm in each instance. We point out that DP_1 algorithm is called a limited number of times with respect to DP_2 (5% of the cases) and mainly in the smallest instances with 1000 items.

Proposed exact approach ($n = 1000$)		Step 1 only	Step 1 and Step 2		Max number of states in DP_2	
Profit class	Weight type	#Opt (%)	Time (%)	Time (%)	Average ($\times 10^3$)	Max ($\times 10^3$)
$p1$	$a1$	72.5	54.0	46.0	0.1	0.2
	$a2$	60.0	56.8	43.2	0.1	0.5
$p2$	$a1$	43.3	50.6	49.4	0.8	16.0
	$a2$	49.2	49.4	50.6	1.6	20.5
$p3$	$a1$	69.2	58.1	41.9	2.8	56.2
	$a2$	52.5	78.0	22.0	1.8	6.8
$p4$	$a1$	48.3	78.0	22.0	2.1	19.6
	$a2$	57.5	73.7	26.3	2.6	14.9
$p5$	$a1$	22.5	43.0	57.0	5.1	36.1
	$a2$	24.2	43.5	56.5	13.4	58.9
$p6$	$a1$	82.5	41.5	58.5	4.9	25.2
	$a2$	27.5	36.0	64.0	7.6	47.8
$p7$	$a1$	59.2	51.0	49.0	0.9	3.2
	$a2$	58.3	48.5	51.5	2.3	7.5

Table 4: Numerical insights of the proposed exact approach for instances with 1000 items.

The results in the tables illustrate the effectiveness of the first step in solving PKP instances. Usually more than 50% of the instances are solved to optimality within this step. When both steps are involved, the computational effort is on average equally distributed. We note however an increase of the percentages of the second step in classes $p5$ and $p6$. The number of states is in general reasonably limited allowing our algorithm to effectively solve all instances considered. The largest values of the number of states (with a maximum of about 3 millions) are reached in the instances with 10000 items.

In the following we also list detailed results for all correlations and capacity ratios.

Proposed exact approach ($n = 10000$)		Step 1 only	Step 1 and Step 2		Max number of states in DP_2	
Profit class	Weight type	#Opt (%)	Time (%)	Time (%)	Average ($\times 10^3$)	Max ($\times 10^3$)
$p1$	$a1$	85.0	79.1	20.9	0.7	6.3
	$a2$	69.2	62.3	37.7	1.5	5.5
$p2$	$a1$	50.0	68.3	31.7	3.4	66.2
	$a2$	52.5	48.6	51.4	19.5	166.1
$p3$	$a1$	77.5	58.0	42.0	146.3	1115.8
	$a2$	56.7	90.1	9.9	45.8	247.4
$p4$	$a1$	73.3	74.5	25.5	119.9	713.0
	$a2$	79.2	83.8	16.2	139.8	885.8
$p5$	$a1$	27.5	39.3	60.7	164.5	1244.3
	$a2$	25.0	39.2	60.8	444.4	3088.3
$p6$	$a1$	83.3	32.0	68.0	187.5	700.5
	$a2$	33.3	26.8	73.2	321.6	1292.2
$p7$	$a1$	55.0	60.3	39.7	81.1	493.3
	$a2$	63.3	60.2	39.8	145.6	764.9

Table 5: Numerical insights of the proposed exact approach for instances with 10000 items.

$n = 1000$			CPLEX 12.5			Algorithm in [1]			Proposed exact approach		
Weight type	τ	Profit class	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
$a1$	0.5	$p1$	0.19	0.25	40	0.00	0.00	40	0.00	0.00	40
		$p2$	0.32	0.92	40	0.00	0.01	40	0.00	0.10	40
		$p3$	10.59	100.00	37	0.10	0.89	40	0.03	0.40	40
		$p4$	62.27	100.00	16	0.14	1.34	40	0.02	0.05	40
		$p5$	11.08	100.00	37	0.09	0.89	40	0.04	0.20	40
		$p6$	2.29	18.94	40	0.00	0.00	40	0.00	0.02	40
		$p7$	36.28	100.00	26	0.00	0.01	40	0.00	0.01	40
	0.1	$p1$	0.20	0.25	40	0.00	0.01	40	0.00	0.00	40
		$p2$	0.49	1.13	40	0.00	0.00	40	0.00	0.05	40
		$p3$	0.54	1.44	40	0.02	0.11	40	0.01	0.04	40
		$p4$	74.13	100.00	11	0.12	1.38	40	0.02	0.18	40
		$p5$	0.93	10.24	40	0.01	0.06	40	0.01	0.05	40
		$p6$	4.23	90.38	40	0.00	0.01	40	0.00	0.03	40
		$p7$	28.61	100.00	29	0.00	0.01	40	0.01	0.02	40
	0.01	$p1$	0.14	0.18	40	0.00	0.01	40	0.00	0.00	40
		$p2$	0.35	1.69	40	0.00	0.00	40	0.00	0.01	40
		$p3$	0.58	8.34	40	0.00	0.03	40	0.00	0.01	40
		$p4$	42.27	100.00	30	0.04	0.20	40	0.01	0.03	40
		$p5$	0.38	3.65	40	0.00	0.01	40	0.00	0.01	40
		$p6$	1.48	4.05	40	0.00	0.06	40	0.00	0.01	40
		$p7$	17.87	100.00	34	0.00	0.01	40	0.00	0.02	40
	0.5	$p1$	0.18	0.32	40	0.00	0.00	40	0.00	0.00	40
		$p2$	0.47	3.19	40	0.00	0.01	40	0.00	0.00	40
		$p3$	10.66	100.00	37	0.89	8.02	40	0.03	0.11	40
		$p4$	53.29	100.00	19	0.50	7.75	40	0.04	0.11	40
		$p5$	23.32	100.00	36	0.65	4.50	40	0.12	1.40	40
		$p6$	4.65	13.57	40	0.00	0.01	40	0.02	0.08	40
		$p7$	17.76	100.00	33	0.00	0.01	40	0.01	0.03	40
	0.1	$p1$	0.22	0.33	40	0.00	0.00	40	0.00	0.00	40
		$p2$	1.52	6.32	40	0.00	0.01	40	0.01	0.27	40
		$p3$	3.40	41.67	40	0.41	2.17	40	0.03	0.29	40
		$p4$	87.51	100.00	6	0.31	7.30	40	0.04	0.26	40
		$p5$	10.75	100.00	38	0.38	1.63	40	0.07	0.38	40
		$p6$	1.91	5.30	40	0.00	0.08	40	0.01	0.04	40
		$p7$	35.62	100.00	26	0.01	0.03	40	0.01	0.07	40
	0.01	$p1$	0.18	0.48	40	0.00	0.01	40	0.00	0.03	40
		$p2$	1.38	4.26	40	0.00	0.01	40	0.00	0.01	40
		$p3$	6.43	18.53	40	0.20	1.17	40	0.02	0.03	40
		$p4$	60.12	100.00	21	0.04	0.14	40	0.03	0.05	40
		$p5$	10.84	60.54	40	0.19	0.84	40	0.02	0.04	40
		$p6$	2.36	6.48	40	0.03	0.15	40	0.01	0.07	40
		$p7$	61.34	100.00	16	0.01	0.06	40	0.02	0.06	40

Table 6: Computational results for instances with 1000 items and different correlations between profits and weights: time (s) and number of optima over 40 instances.

$n = 10000$			CPLEX 12.5			Algorithm in [1]			Proposed exact approach		
Weight type	τ	Profit class	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
a1	0.5	p1	1.00	2.28	40	0.01	0.02	40	0.02	0.03	40
		p2	3.73	9.40	40	0.01	0.02	40	0.02	0.03	40
		p3	62.62	100.00	16	45.29	100.00	23	6.65	58.46	40
		p4	82.03	100.00	8	17.43	100.00	34	2.79	10.73	40
		p5	53.44	100.00	20	24.99	100.00	33	7.16	70.71	40
		p6	90.65	100.00	12	0.01	0.06	40	0.23	2.56	40
		p7	21.31	100.00	33	0.09	0.33	40	0.15	0.51	40
	0.1	p1	1.12	2.68	40	0.01	0.01	40	0.01	0.02	40
		p2	11.05	33.84	40	0.01	0.02	40	0.02	0.13	40
		p3	55.29	100.00	21	26.62	100.00	33	1.02	8.73	40
		p4	95.24	100.00	2	16.64	100.00	37	3.54	18.88	40
		p5	20.70	100.00	35	6.24	78.90	40	0.76	6.13	40
		p6	77.25	100.00	24	0.09	1.97	40	0.33	5.38	40
		p7	64.61	100.00	15	0.27	0.94	40	0.57	2.26	40
	0.01	p1	0.73	1.28	40	0.00	0.01	40	0.00	0.01	40
		p2	1.45	2.74	40	0.00	0.02	40	0.01	0.02	40
		p3	15.92	100.00	37	4.86	59.95	40	0.12	1.00	40
		p4	97.53	100.00	1	18.97	100.00	35	2.10	14.20	40
		p5	2.22	6.56	40	0.86	7.79	40	0.07	0.56	40
		p6	81.35	100.00	22	1.91	40.17	40	0.11	1.69	40
		p7	67.09	100.00	15	0.14	0.56	40	0.54	2.50	40
	0.5	p1	1.93	11.22	40	0.01	0.02	40	0.02	0.02	40
		p2	3.86	11.72	40	0.01	0.02	40	0.02	0.03	40
		p3	70.77	100.00	18	53.23	100.00	23	8.09	25.55	40
		p4	86.05	100.00	6	24.86	100.00	35	8.72	26.06	40
		p5	60.26	100.00	19	24.30	100.00	32	9.93	58.60	40
		p6	86.29	100.00	7	0.08	0.95	40	2.62	16.65	40
		p7	14.30	100.00	36	0.31	2.23	40	0.48	3.34	40
	0.1	p1	1.27	2.19	40	0.01	0.02	40	0.01	0.02	40
		p2	10.99	29.04	40	0.01	0.02	40	0.04	0.66	40
		p3	66.31	100.00	19	50.95	100.00	24	4.33	14.29	40
		p4	98.01	100.00	1	34.25	100.00	35	11.28	80.02	40
		p5	54.25	100.00	21	27.97	100.00	35	5.97	40.59	40
		p6	82.95	100.00	13	0.29	1.47	40	2.28	7.87	40
		p7	51.33	100.00	20	0.77	3.94	40	1.83	11.46	40
	0.01	p1	9.37	100.00	37	0.01	0.04	40	0.01	0.03	40
		p2	22.45	100.00	34	0.01	0.07	40	0.06	0.73	40
		p3	85.32	100.00	9	40.59	100.00	27	4.18	29.00	40
		p4	100.00	100.00	0	19.92	100.00	36	3.46	15.36	40
		p5	83.47	100.00	8	18.96	100.00	34	5.22	20.51	40
		p6	73.90	100.00	24	10.50	100.00	39	0.96	3.90	40
		p7	54.74	100.00	19	0.55	3.47	40	1.92	8.74	40

Table 7: Computational results for instances with 10000 items and different correlations between profits and weights: time (s) and number of optima over 40 instances.

$n = 1000$			CPLEX 12.5			Algorithm in [1]			Proposed exact approach		
Weight type	τ	Penalty class	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
a1	0.5	π_1	6.65	100.00	33	0.01	0.05	35	0.01	0.03	35
		π_2	16.33	100.00	30	0.02	0.14	35	0.01	0.12	35
		π_3	19.56	100.00	29	0.08	0.89	35	0.02	0.14	35
		π_4	21.98	100.00	28	0.03	0.39	35	0.01	0.06	35
		π_5	22.95	100.00	28	0.05	0.38	35	0.01	0.10	35
		π_6	25.89	100.00	27	0.04	0.89	35	0.01	0.05	35
		π_7	15.49	100.00	30	0.09	1.34	35	0.01	0.03	35
		π_8	11.73	100.00	31	0.06	0.63	35	0.03	0.40	35
	0.1	π_1	7.53	100.00	33	0.00	0.05	35	0.01	0.18	35
		π_2	15.01	100.00	30	0.01	0.08	35	0.01	0.08	35
		π_3	29.18	100.00	25	0.01	0.08	35	0.01	0.03	35
		π_4	17.96	100.00	29	0.01	0.06	35	0.01	0.03	35
		π_5	18.44	100.00	29	0.01	0.09	35	0.01	0.03	35
		π_6	12.68	100.00	31	0.01	0.11	35	0.00	0.04	35
		π_7	17.93	100.00	30	0.09	1.38	35	0.01	0.05	35
		π_8	5.98	100.00	33	0.03	0.58	35	0.01	0.05	35
	0.01	π_1	0.70	3.83	35	0.00	0.00	35	0.00	0.01	35
		π_2	7.19	100.00	33	0.00	0.03	35	0.01	0.03	35
		π_3	7.52	100.00	34	0.00	0.05	35	0.00	0.03	35
		π_4	20.09	100.00	30	0.01	0.05	35	0.01	0.03	35
		π_5	16.57	100.00	32	0.01	0.05	35	0.00	0.03	35
		π_6	3.37	30.16	35	0.00	0.01	35	0.00	0.01	35
		π_7	15.94	100.00	30	0.02	0.20	35	0.00	0.01	35
		π_8	0.70	8.34	35	0.01	0.09	35	0.00	0.01	35
a2	0.5	π_1	10.77	100.00	32	0.02	0.14	35	0.02	0.17	35
		π_2	16.60	100.00	31	0.08	1.07	35	0.07	1.40	35
		π_3	22.18	100.00	29	0.42	4.57	35	0.04	0.32	35
		π_4	19.38	100.00	30	0.51	7.81	35	0.03	0.21	35
		π_5	21.57	100.00	29	0.28	3.00	35	0.03	0.28	35
		π_6	11.32	100.00	32	0.47	8.02	35	0.03	0.23	35
		π_7	9.68	100.00	32	0.49	7.75	35	0.02	0.11	35
		π_8	14.59	100.00	30	0.05	0.80	35	0.01	0.09	35
	0.1	π_1	14.78	100.00	32	0.02	0.17	35	0.05	0.38	35
		π_2	21.28	100.00	28	0.04	0.30	35	0.02	0.22	35
		π_3	18.82	100.00	29	0.22	2.16	35	0.03	0.27	35
		π_4	30.67	100.00	25	0.28	2.17	35	0.03	0.13	35
		π_5	25.16	100.00	27	0.22	1.98	35	0.02	0.12	35
		π_6	28.61	100.00	26	0.19	1.63	35	0.02	0.09	35
		π_7	10.08	100.00	32	0.26	7.30	35	0.02	0.11	35
		π_8	11.68	100.00	31	0.04	0.44	35	0.01	0.07	35
	0.01	π_1	1.60	12.74	35	0.00	0.01	35	0.01	0.03	35
		π_2	28.11	100.00	28	0.01	0.14	35	0.01	0.04	35
		π_3	26.29	100.00	28	0.15	0.73	35	0.02	0.05	35
		π_4	19.68	100.00	31	0.11	1.17	35	0.02	0.05	35
		π_5	33.97	100.00	25	0.10	0.67	35	0.02	0.05	35
		π_6	24.26	100.00	29	0.12	0.84	35	0.02	0.06	35
		π_7	17.34	100.00	30	0.02	0.10	35	0.01	0.03	35
		π_8	11.77	100.00	31	0.02	0.16	35	0.00	0.02	35

Table 8: Computational results for instances with 1000 items and different correlations between penalties and weights: time (s) and number of optima over 35 instances.

$n = 10000$			CPLEX 12.5			Algorithm in [1]			Proposed exact approach		
Weight type	τ	Penalty class	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
a1	0.5	π_1	26.96	100.00	29	1.07	8.74	35	1.04	10.54	35
		π_2	51.68	100.00	18	4.93	100.00	34	0.89	8.74	35
		π_3	45.50	100.00	20	8.83	100.00	33	1.06	5.78	35
		π_4	52.72	100.00	17	24.28	100.00	28	2.22	17.67	35
		π_5	54.65	100.00	17	18.01	100.00	29	1.66	10.09	35
		π_6	55.55	100.00	16	23.99	100.00	27	2.46	20.65	35
		π_7	38.21	100.00	25	11.72	100.00	31	1.08	10.73	35
		π_8	34.49	100.00	27	7.54	100.00	33	9.05	70.71	35
	0.1	π_1	27.19	100.00	28	0.88	18.70	35	0.72	18.88	35
		π_2	48.67	100.00	22	2.17	20.73	35	0.70	6.27	35
		π_3	54.65	100.00	18	7.52	100.00	33	1.30	18.19	35
		π_4	59.08	100.00	17	15.20	100.00	32	0.91	7.16	35
		π_5	57.49	100.00	19	8.46	100.00	34	1.00	8.53	35
		π_6	51.93	100.00	21	7.55	100.00	34	0.39	2.26	35
		π_7	48.77	100.00	22	11.15	100.00	32	0.85	8.59	35
		π_8	23.95	100.00	30	4.07	73.67	35	1.27	8.73	35
	0.01	π_1	21.36	100.00	30	0.17	3.00	35	0.31	6.57	35
		π_2	37.31	100.00	26	6.20	73.05	35	0.58	5.66	35
		π_3	45.55	100.00	21	2.26	33.15	35	0.28	3.69	35
		π_4	45.75	100.00	21	2.36	59.95	35	0.33	2.50	35
		π_5	49.01	100.00	21	1.25	10.15	35	0.46	5.62	35
		π_6	44.44	100.00	23	1.28	26.15	35	0.58	14.20	35
		π_7	38.94	100.00	22	13.75	100.00	31	0.47	3.36	35
		π_8	21.96	100.00	31	3.30	100.00	34	0.35	6.39	35
a2	0.5	π_1	50.50	100.00	18	4.70	44.55	35	2.91	17.61	35
		π_2	51.30	100.00	18	16.08	100.00	30	4.55	53.64	35
		π_3	46.81	100.00	21	19.17	100.00	29	5.14	42.97	35
		π_4	55.72	100.00	19	14.48	100.00	32	4.80	58.60	35
		π_5	52.02	100.00	19	20.40	100.00	30	4.84	29.21	35
		π_6	47.85	100.00	21	18.55	100.00	29	4.29	39.81	35
		π_7	37.68	100.00	24	11.97	100.00	32	3.61	25.55	35
		π_8	27.78	100.00	26	12.14	100.00	33	4.02	26.06	35
	0.1	π_1	46.84	100.00	21	11.89	100.00	33	3.47	41.73	35
		π_2	49.39	100.00	21	15.23	100.00	34	3.55	40.59	35
		π_3	57.48	100.00	18	21.44	100.00	29	4.78	28.76	35
		π_4	60.68	100.00	18	19.71	100.00	31	3.62	19.90	35
		π_5	73.48	100.00	10	23.79	100.00	29	4.05	14.26	35
		π_6	57.68	100.00	18	23.00	100.00	31	6.56	80.02	35
		π_7	41.52	100.00	22	8.76	100.00	33	1.70	7.87	35
		π_8	30.19	100.00	26	6.75	100.00	34	1.70	15.50	35
	0.01	π_1	61.19	100.00	16	2.94	26.59	35	3.98	29.00	35
		π_2	60.12	100.00	19	5.40	49.36	35	3.37	20.51	35
		π_3	65.69	100.00	14	15.40	100.00	31	1.87	12.09	35
		π_4	77.25	100.00	10	16.77	100.00	30	2.39	13.24	35
		π_5	73.10	100.00	12	17.29	100.00	30	2.08	11.25	35
		π_6	60.01	100.00	18	16.08	100.00	30	1.71	8.97	35
		π_7	58.23	100.00	17	24.62	100.00	30	1.70	6.82	35
		π_8	34.99	100.00	25	4.95	51.47	35	0.97	6.53	35

Table 9: Computational results for instances with 10000 items and different correlations between penalties and weights: time (s) and number of optima over 35 instances.

6. Approximation results

In this section we investigate the approximability of PKP. The classical 0–1 Knapsack Problem admits fully polynomial time approximation schemes (FPTAS), see, e.g. [3]. PKP has “only” an additional penalty to consider in the objective with respect to KP. Thus, one might expect some straightforward approximation algorithm for this problem as well. Nonetheless, we prove here the general result that no polynomial time approximation algorithm exists for PKP (under $\mathcal{P} \neq \mathcal{NP}$).

Theorem 11. *PKP does not admit a polynomial time algorithm with a bounded approximation ratio unless $\mathcal{P} = \mathcal{NP}$.*

Proof. The theorem is proved by reduction from the Subset Sum Problem (SSP). Given n items with integer weights w'_j ($j = 1, \dots, n$) and a value W' (with

$\sum_{j=1}^n w'_j > W'$), we recall that the decision version of SSP is an NP-complete problem and asks whether there exists a subset of items represented by x^* such that $\sum_{j=1}^n w'_j x_j^* = W'$.

We build an instance of PKP with n items, profits and weights $p_j = w_j = w'_j$, penalties $\pi_j = W' - 1$ ($j = 1, \dots, n$) and capacity $c = W'$. The capacity constraint implies that for every feasible solution there is $\sum_{j=1}^n p_j x_j = \sum_{j=1}^n w_j x_j \leq W'$. The penalty value will be equal to either $W' - 1$ if we pack at least one item or 0 otherwise, therefore the optimal solution of this PKP instance is bounded by $\sum_{j=1}^n p_j x_j - (W' - 1) \leq 1$. Not placing any item in the knapsack attains the trivial solution with value equal to 0. By integrality of the input data, this limits the optimal solution value to 0 or 1, where the latter value can be reached if and only if the Subset Sum Problem has a solution.

Hence, if there was a polynomial time algorithm for PKP with a bounded approximation ratio, we could decide SSP just by checking if the approximate solution of PKP is strictly positive. Clearly this is not possible unless $\mathcal{P} = \mathcal{NP}$. \square

While the result of Theorem 11 rules out any reasonable approximation for the general case, one can impose mild restrictions on the input data which still permit fully polynomial time approximation algorithms. All our results are based on the following simple approximation algorithm: Similar to the exact algorithm sketched in Section 2 we consider all n choices for the penalty value, namely $\Pi \in \{\pi_1, \dots, \pi_n\}$. For each choice j of the leading item with $\Pi = \pi_j$, we compute a suboptimal solution of problem PKP_j^+ by packing item j into the knapsack and applying a $(1 - \delta)$ -approximation algorithm for the remaining knapsack problem with capacity $c - w_j$ and item set $\{j + 1, \dots, n\}$. The optimal solution value of the latter problem will be denoted as z_j^R .

As an output of the resulting approximation algorithm $A(\delta)$ with objective function value $z^A(\delta)$ we use the maximum value obtained over all n iterations (including the empty set). For a constant $\delta > 0$, algorithm $A(\delta)$ can be performed by running n times an FPTAS for KP. Note that if $z^* = 0$, then also $A(\delta)$ will output a value of 0. Thus we can assume by integrality of the input data:

$$z^* = p_{j^*} + z_{j^*}^R - \pi_{j^*} \geq 1. \quad (31)$$

As a general bound on the performance of $A(\delta)$ we get:

$$\begin{aligned} z^A(\delta) &\geq p_{j^*} + (1 - \delta)z_{j^*}^R - \pi_{j^*} \\ &= (1 - \varepsilon)z^* + (\varepsilon - \delta)z_{j^*}^R - \varepsilon(\pi_{j^*} - p_{j^*}) \end{aligned}$$

Hence, we obtain an FPTAS for a suitable choice of $\delta \leq \varepsilon$ if we can prove:

$$(\varepsilon - \delta)z_{j^*}^R \geq \varepsilon(\pi_{j^*} - p_{j^*}) \quad (32)$$

Note that whenever inequality $p_{j^*} \geq \pi_{j^*}$ is implied, condition (32) is trivially satisfied for any $\delta \leq \varepsilon$.

In the following we will describe four relevant cases which all permit an FPTAS

for PKP. We start with the case where each item has a profit greater than (or equal to) its penalty value.

Proposition 12. *If $p_j \geq \pi_j$ for $j = 1, \dots, n$, then algorithm $A(\delta)$ is an FPTAS for PKP.*

Proof. It is trivial to see that setting $\delta := \varepsilon$ the right-hand side of (32) is always less than (or equal to) zero for every j and thus the inequality is always fulfilled. \square

We henceforth assume the restricting condition $\pi_{j^*} - p_{j^*} \geq 0$. We first consider the case where penalties are bounded by a given constant C .

Proposition 13. *If $\pi_j + 1 \leq C$ for $j = 1, \dots, n$ and a constant C , algorithm $A(\delta)$ constitutes an FPTAS.*

Proof. We can choose $\delta := \frac{\varepsilon}{C}$ and consider from (31) that $z_{j^*}^R \geq \pi_{j^*} - p_{j^*} + 1$. We have that:

$$(\varepsilon - \delta)z_{j^*}^R \geq (\varepsilon - \delta)(\pi_{j^*} - p_{j^*} + 1) \quad (33)$$

$$= \varepsilon(\pi_{j^*} - p_{j^*} + 1) - \frac{\varepsilon}{C}(\pi_{j^*} - p_{j^*} + 1) \quad (34)$$

$$\geq \varepsilon(\pi_{j^*} - p_{j^*}) + \varepsilon - \frac{\varepsilon}{C}(C - p_{j^*} + 1) \quad (35)$$

$$= \varepsilon(\pi_{j^*} - p_{j^*}) + \frac{\varepsilon}{C}(p_{j^*} - 1) \quad (36)$$

$$\geq \varepsilon(\pi_{j^*} - p_{j^*}) \quad (37)$$

The last inequality (37) follows from the integrality of profits. Hence, condition (32) is shown. \square

As generalization of the case in Proposition 12, we consider for each item smaller profits than penalties and manage to derive an FPTAS as long as this difference is bounded by a constant.

Proposition 14. *PKP admits an FPTAS if $\pi_j - p_j \leq C$ for $j = 1, \dots, n$ and a constant C .*

Proof. By choosing $\delta := \frac{\varepsilon}{C+1}$ we get:

$$(\varepsilon - \delta)z_{j^*}^R = \left(\varepsilon - \frac{\varepsilon}{C+1}\right)z_{j^*}^R \quad (38)$$

$$\geq \left(\varepsilon - \frac{\varepsilon}{\pi_{j^*} - p_{j^*} + 1}\right)z_{j^*}^R \quad (39)$$

$$\geq \left(\varepsilon - \frac{\varepsilon}{\pi_{j^*} - p_{j^*} + 1}\right)(\pi_{j^*} - p_{j^*} + 1) \quad (40)$$

$$= \varepsilon(\pi_{j^*} - p_{j^*}) + \varepsilon - \varepsilon \quad (41)$$

For inequality (40) we invoke again (31). This shows condition (32). \square

Finally, consider the case where there exists a bounded interval containing all profit and penalty values. This can be expressed by assuming a constant parameter $\rho \in (1/2, 1)$ with $p_{\min} \geq \rho \cdot \pi_{\max}$, i.e. all profits are not too small compared to the largest penalty. On the other hand, profits can well be arbitrarily large. The following proposition holds.

Proposition 15. *There is an FPTAS for PKP if $p_{\min} \geq \rho \cdot \pi_{\max}$ with $\rho \in (1/2, 1)$.*

Proof. If the optimal solution consists only of the leading item, then $A(\delta)$ also yields the optimum. Thus, we have $z_{j^*}^R \geq p_{\min} \geq \rho \pi_{\max}$. Choosing $\delta := \varepsilon \frac{2\rho-1}{\rho}$ (note that $\delta > 0$ for $\rho > 1/2$) we get:

$$(\varepsilon - \delta)z_{j^*}^R \geq (\varepsilon - \delta)\rho \pi_{\max} \quad (42)$$

$$= \left(\varepsilon - \varepsilon \frac{2\rho-1}{\rho}\right)\rho \pi_{\max} \quad (43)$$

$$= (\varepsilon\rho - 2\varepsilon\rho + \varepsilon)\pi_{\max} \quad (44)$$

$$= \varepsilon(1 - \rho)\pi_{\max} \quad (45)$$

$$\geq \varepsilon(\pi_{\max} - p_{\min}) \quad (46)$$

$$\geq \varepsilon(\pi_{j^*} - p_{j^*}) \quad (47)$$

This shows again condition (32). \square

We remark that, although we cannot exclude other approximation schemes for PKP, it seems hard to construct any meaningful approximation algorithm different from $A(\delta)$. The algorithm considers each term $p_j - \pi_j$ explicitly and thus will also include the part of the optimal solution value contributed by the leading item $p_{j^*} - \pi_{j^*}$. Then, the knapsack problem $z_{j^*}^R$ is solved by a $(1 - \delta)$ -approximation with a suitably chosen parameter δ which is the best one can do for the remaining sub-problem.

7. Conclusions

We proposed a dynamic programming based exact approach for PKP which leverages an algorithmic framework originally constructed for KP. The proposed approach turns out to be very effective in solving instances of the problem with up to 10000 items and favorably compares to both solver CPLEX 12.5 and an exact algorithm in the literature. From a theoretical point of view we also show that PKP can be solved in the same pseudopolynomial running time $O(nc)$ as the standard knapsack problem. We also gave further insights on the structure and properties of PKP by providing a characterization of its linear relaxation and an effective procedure to compute upper bounds on the problem. By studying the approximability of PKP, we showed rather surprisingly that there is no polynomial time approximation algorithm with bounded approximation ratio, while imposing some mild conditions on the input of PKP allows an FPTAS. In future research, we will investigate extensions of our procedures to other KP

generalizations. It would also be interesting to evaluate the performances of our approach on new benchmark and challenging PKP instances.

Acknowledgments

We thank the authors of [1] for providing us with the code of their algorithm and the generation scheme of the instances.

Rosario Scatamacchia was supported by a fellowship from TIM Joint Open Lab SWARM (Turin, Italy). Ulrich Pferschy was supported by the University of Graz project "Choice-Selection-Decision".

References

- [1] A. Ceselli and G. Righini. An optimization algorithm for a penalized knapsack problem. *Operations Research Letters*, 34:394–404, 2006.
- [2] R. S. Dembo and P. L. Hammer. A reduction algorithm for knapsack problems. *Methods of Operations Research*, 36:49–60, 1980.
- [3] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.
- [4] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: a survey. *European Journal of Operational Research*, 141:241–252, 2002.
- [5] S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0–1 knapsack problems. *European Journal of Operational Research*, 123:325–332, 2000.
- [6] S. Martello and P. Toth. A new algorithm for the 0–1 knapsack problem. *Management Science*, 34:633–644, 1988.
- [7] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, 1990.
- [8] D. Pisinger. A minimal algorithm for the 0–1 knapsack problem. *Operations Research*, 45:758–767, 1997.
- [9] D. Pisinger. Linear time algorithms for knapsack problems with bounded weights. *Journal of Algorithms*, 33:1–14, 1999.